Supporting Reconfigurable Parallel Multimedia Applications

Maik Nijhuis, Herbert Bos, and Henri E. Bal

Department of Computer Science, Vrije Universiteit, Amsterdam
{maik,herbertb,bal}@cs.vu.nl

Abstract. Programming multimedia applications for System-on-Chip (SoC) architectures is difficult because streaming communication, user event handling, reconfiguration, and parallelism have to be dealt with. We present Hinch, a runtime system for multimedia applications, that efficiently exploits parallelism by running the application in a dataflow style. The application has to be implemented as components that communicate using streams. Reconfigurability is supported by a generic component interface. Measurements have been performed on a SpaceCake SoC architecture simulator. Hinch can easily be ported to other shared-memory architectures.

1 Introduction

The problem we address in this paper is the complexity of programming embedded System-on-Chip (SoC) architectures with multiple processing units operating in parallel. These architectures are becoming more and more popular in the field of consumer electronics, especially in the area of multimedia applications. This trend, which is already visible, is likely to become even more important in the future for several reasons. First, media applications increasingly require complex processing, for example new coding algorithms and dynamic picture-in-picture. Second, the processing is applied to increasing amounts of data per time-unit, such as multichannel HDTV. Third, media applications often exhibit much potential parallelism. Fourth, hardware vendors have already opted for multi-core processors as the key to speed, partly because it is hard to crank up clock speeds at the same rate as in the past (e.g., Cell[1], Network processors[2], Xeon, and Opteron). Unfortunately, programming such parallel hardware is challenging and very much an open research problem.

In this paper, we present a solution that facilitates application development for SoC architectures. While our target domain includes a broad range of applications, we focus our examples and discussion on TV sets, for ease of explanation. The SpaceCake SoC architecture[3], developed by Philips, is used as the experimentation platform. For testing purposes, the applications can also run natively on Linux.

By careful observation of several applications, we obtained a core set of properties that a multimedia run time system should cater for. We used these properties to derive requirements for our system. Our system, named Hinch, exploits the following properties of typical multimedia applications:

1. The application consists of several kernels that perform a specific operation, such as motion estimation. To manage large numbers of kernels, we should be able to group them into *components*, which in turn can be grouped into higher-level components. The resulting application will be organized hierarchically like a tree with kernels at the leaf nodes.
2. The configuration of kernels changes as a result of asynchronous user inputs and other events. For example, by pressing a button a user may add a picture-in-picture to the television screen. Hinch supports events and allows the kernel configuration to change at run time. As we typically do not want to stop the complete application as reconfiguration occurs, we allow subtrees in the tree-based hierarchy to be reconfigured without interfering with user experience.
3. Individual nodes in the tree communicate either via streaming channels (e.g., a motion estimator kernel calculates motion vectors that are used by a motion compensated de-interlacer kernel), or via events (e.g., a component sends an event that a sub-program has started). Hinch supports both streaming and events.
4. Multimedia applications exhibit both task- and data-parallelism. Hinch is able to map the tree-based component hierarchy on hardware such that both forms of parallelism are exploited. For task parallelism this implies that different kernels are mapped on different functional units. For data parallelism, multiple instances of a kernel have to run concurrently.

While it is well known how to support the individual properties, to the best of our knowledge, Hinch is the first system that supports all, while greatly simplifying the SoC programmers' task. Moreover, measurements show Hinch incurs only little overhead and achieves a parallelization efficiency of about 95 % with 9 processors. The major difficulties we encounter are combining task- and data-parallelism[4], supporting dynamic reconfiguration and handling asynchronous events. Hinch can be used as a lower layer in a programming environment for building multimedia applications. We plan to combine Hinch with higher level layers such as SPC-XML[5].

The remainder of this paper is organized as follows. In Section 2 we will explain reconfigurability requirements. In Section 3 we will describe the design of Hinch. In Section 4 we will show the results of using Hinch on the SpaceCake architecture. Related work is discussed in Section 5. Finally, the paper is concluded in Section 6.

2 Reconfigurability

Many multimedia applications need support for reconfiguration. This can be due to user input (e.g., the user wants to add a picture-in-picture), or to available resources (e.g., scaling down quality when less bandwidth is available). Reconfiguration can be performed by adjusting parameters of application components (component reconfiguration) or by adding and removing components while the application is running (application reconfiguration). In this section, we give two examples of reconfigurable applications. Both applications are used for the experiments described in Section 4.

2.1 Add/Remove Components

In a dynamic Picture-In-Picture (PiP) application, the user can add or remove small subpictures (of different TV channels) on the screen. The structure of this application is shown in Fig. 1. The downscale components reduce the size of their inputs and the blend component merges the downscaled images into the main background image. When a picture is added, input and downscale components are created and connected to the blender. The blender is then notified that it has to blend one more picture into its output. When a picture is deleted, the blender is notified, the connection to the blender is removed and the picture-in-picture input and downscale components are destroyed. The notifications to the blender are an example of component reconfiguration.

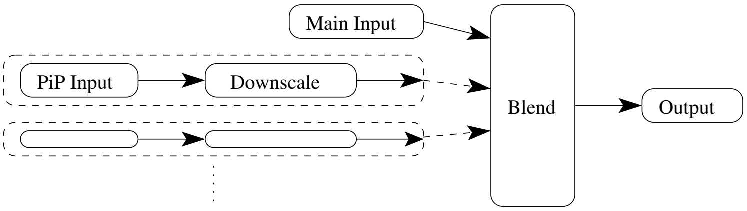

Fig. 1. Picture-in-Picture application

2.2 Replace Components

Temporal upscaling (increasing the frame rate) of a movie can be done at different quality levels that have different computation requirements. A trivial temporal upscaler simply copies existing images to create new images. An advanced temporal up scaler can perform motion estimation and use the motion vectors to compute the new images.

We have created an application ('Tups') that performs temporal upscaling of an image sequence by a given factor. Copy mode and motion estimation mode are both supported. The application layout for both modes is shown in Fig. 2. The application can dynamically switch between the two modes by replacing the middle component, for instance depending on available resources.

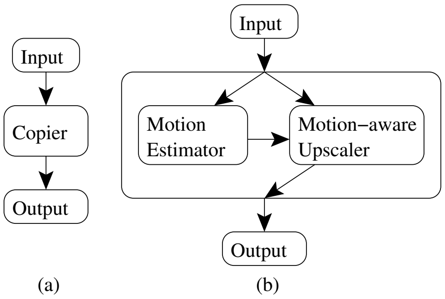

Fig. 2. Tups application: copy mode (a) and motion estimation mode (b)

3 Run Time System

In this section we explain the design of Hinch and the model for applications that use it. Hinch, as well as the applications that use it, are written in C. Hinch consists of several modules that provide functionality to the application and/or to other Hinch modules. These modules are also described below.

3.1 Application Layout

A Hinch application consists of components that are actors in a dataflow process network[6]. The application is run by executing *iterations* of the dataflow graph, in which each actor is fired one or more times. One firing corresponds to running one iteration of the component. A graph iteration begins by scheduling the initial component(s). The other components are scheduled as soon as their predecessors in the dataflow graph have finished. There is no restriction on the shape of the dataflow graph. For a video processing application, the components typically contain image processing kernels. One iteration then consists of processing one image frame from the video stream.

Dataflow graphs can be nested using special grouping components. Grouping components contain child components and connect these into a dataflow graph. When an iteration of a grouping component is run, an iteration of the dataflow graph inside the grouping component is run. When this ’inner’ iteration has finished, the successors of the grouping component in the higher level dataflow graph are scheduled.

All components have a generic interface, which provides an abstraction of the component. This interface contains functions to create, configure and destroy instances of the component, to get its properties, and to run an iteration of the component. A grouping component has extra functions to add, remove, replace, and (dis)connect its contained children. All functions except the run function are used to configure the application, at the start of the application. An application can also be reconfigured at run time using these functions. The connections between children correspond to dataflow dependencies. Connections can also include a data stream, as will be explained in the next subsection.

A central job queue in shared memory is used to accomplish automatic load balancing. To enable usage of Hinch on distributed memory machines, we plan to scale the job queue to a distributed version using the algorithms of [7]. A parallel program consists of multiple threads that continuously execute jobs from the queue, and add new jobs to the queue that are ready to run. We avoid expensive context switches by running a single thread per processor. A job in Hinch consists of running an iteration of a component. When a job has finished, the dataflow graph is used to find the successors of this job. These are added to the job queue if they are ready to be run.

To build a parallel program, the programmer only has to build components, specify the connections between these components, and call some initialization routines. Using the job queue, Hinch makes sure that the program runs in parallel. We plan to add an XML layer on top of Hinch for specifying the component connections. This XML layer can then also be used for performance prediction using PAM-SoC[8].

The Hinch application model resembles that of Koala[9]. A Hinch component iteration corresponds to the execution of a Koala task. As in Koala, Hinch components can only be coupled if their interfaces match and components can be grouped recursively.

3.2 Streams

Components can communicate in a streaming fashion using a stream module. This modules provides similar streaming functions as the StreamIt[10] language and a communication abstraction similar to Space-Time Memory[11]. A component does not know with whom it is communicating. It merely has to read and write the appropriate streams, which are parameters to the `run` function. This way, a component can easily be reused in another part of the application or in a different application.

Streams are implemented using an efficient zero-copy protocol. The producer can write to the stream after allocating a write buffer. When it is finished writing, it commits the buffer. The consumer can then read the data from the same buffer. The zero-copy protocol is only possible at shared memory machines. For distributed memory machines, a different implementation must be used.

Multicast streams are also supported by Hinch. These streams are shared by multiple dataflow connections. There is also support for reading a fixed amount of old data, which has been used in previous iterations, from the stream. This is similar to the peek functionality in StreamIt[10].

3.3 Task Parallelism

Task parallelism is supported by Hinch in two ways, as shown in Fig. 3. The circles indicate components, and the arrows indicate connections between them. We assume one iteration of the application processes one image frame using multiple image processing components. The first type of task parallelism is running multiple iterations concurrently in a pipeline style. When component A has finished frame 1, A could be processing frame 2 while B is processing frame 1. The second type of task parallelism is running independent tasks concurrently. Components V and W have no dependency so they can be concurrently active in the same iteration. Both types of task parallelism can be combined. For example, if V and W both have finished frame 2 they can start processing frame 3 while X is processing frame 2.

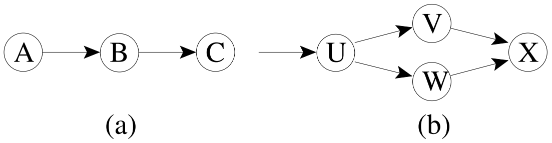

Fig. 3. Task parallelism. Pipeline-style (a) and independent tasks (b)

3.4 Data Parallelism

Normally, a component iteration processes the data it gets by reading its input streams once (say one image frame). However, often this processing can be done in parallel, in which case a component iteration consists of processing a *slice* (multiple lines) instead of a frame. Hinch has slicing helper functions for components, which contain common code needed to code slicing. These functions tell the component which slice of which frame is to be processed, do the appropriate stream reading and writing, and handle out-of-order execution of sliced iterations. By using these functions, exploiting data parallelism becomes easy.

One might argue that data parallelism can also be obtained by reducing the stream granularity from a frame to a slice. However, processing a slice usually requires the data in the previous and next slice for the pixels at the boundary, for example with convolution kernels. Programming a component is more difficult in this case because boundary conditions have to be dealt with. In our approach, this is not necessary since the whole frame is in a continuous memory area.

3.5 Reconfiguration and Event Handling

Reconfigurability is supported by the general component interface. Components can be dynamically created, destroyed, grouped, and connected at run time. To avoid race conditions, the application parts that are to be reconfigured are made idle before reconfigurating.

The replace function in the grouping component interface can replace a child component by a component that has the same I/O interface. Without the replace function, this has to be accomplished by removing all connections to the old child, removing the child, adding the new child, and creating the connections to the new child. With the replace function, removing and creating the connections is not necessary because the new child has the same interface.

Asynchronous events can easily be handled by buffering them in an event queue, as shown in Fig. 4. The event queue is periodically emptied by the manager component, which is run at the end of every iteration. It regulates the number of concurrent active iterations in the application using the control flow connection to the start of the application. The manager component can halt the program for reconfiguration by lowering the number of active iterations to zero.

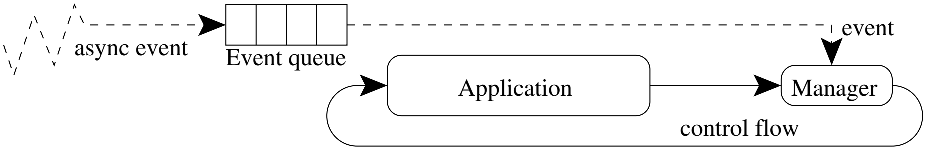

Fig. 4. Event queue (Application overview)

4 Experiments

To verify the usefulness of Hinch, we measured parallelization efficiency and reconfigurability overhead for the picture-in-picture (PiP) and temporal up scaler (Tups) applications. The temporal up scaler doubles the amount of image frames in its input stream by inserting a new image frame after each image. All measurements were done using 720x576 video files. I/O (reading the files and writing the final result) is not included in the measurements.

Since SpaceCake[3] hardware is not yet available, all experiments are run using simulation software, which simulates a tile with multiple TriMedia cores. A TriMedia is a VLIW processor aimed at multimedia applications. At a tile, each TriMedia has its own level 1 cache. The level 2 cache is shared between all TriMedias. The SpaceCake architecture allows multiple tiles to be combined. We plan to add support for multiple tiles in the future.

The TriMedia cycle counter is used for all measurements. For the experiments on a single processor, we turned off all inter-processor synchronization, e.g., locking of shared variables. All other measurements use a parallel version. The number of slices is set to 4 for the picture-in-picture applications and 9 for the temporal up scaling applications. These settings yielded the best results.

4.1 Parallelism

Figure 5 shows the speedup of processing 96 image frames with four variants of the PiP application. These variants (PiP-0, PiP-1, PiP-2 and PiP-3) process zero, one, two, and three pictures-in-picture, respectively. PiP-0 does not exhibit much speedup because it is a trivial application that merely copies its input to its output. However, the speedup stays constant when run at a larger number of nodes, which shows that the overhead of Hinch does not increase with the number of nodes. PiP-1 reaches maximum speedup at 8 nodes. There is no more parallelism to exploit when PiP-1 is run at 9 nodes.

Figure 6 shows the speedup of the Tups application in copy mode (tups-copy), motion estimation mode (tups-me), and in a reconfigurable mode (tups-reconf). Tups-copy creates the newly inserted images by copying the previous image. Tups-me generates the newly inserted images from the two adjacent images using motion estimation techniques. Tups-reconf is a mixture of tups-copy and tups-me and will be explained in the next subsection. These applications process 68 image frames. Again, the trivial application (tups-copy) does not exhibit a good speedup and its speedup stays constant when run at a larger number of nodes.

Both Fig. 5 and Fig. 6 show that Hinch provides the means to efficiently parallelize these multimedia applications. At nine nodes, the efficiency of PiP-2 and tups-me are 94,2 % and 95,8 %, respectively.

4.2 Reconfigurability

We have created four reconfigurable applications. Three of these are variants of the PiP application, the other is a variant of the Tups application. These applications process an equal amount of frames as their non-reconfigurable counterparts. The four variants are:

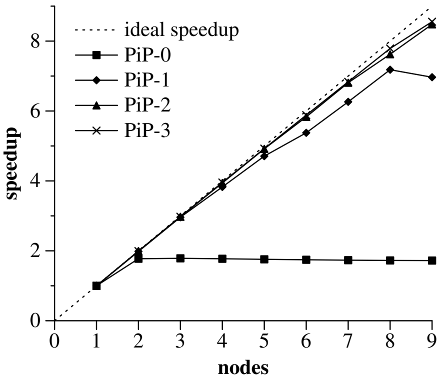

Fig. 5. PiP application speedup

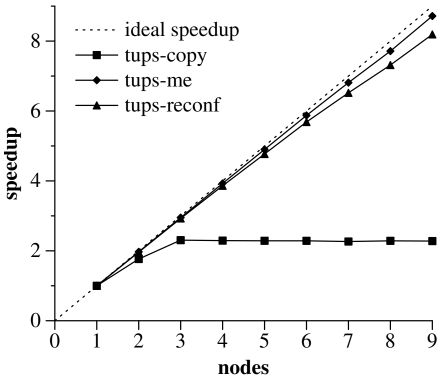

Fig. 6. Tups application speedup

1. PiP-01. Half of the frames have no pictures-in-picture, the other half have one picture-in-picture.
2. PiP-12. Half of the frames has one picture-in-picture, the other half has two pictures-in-picture.
3. PiP-012. One third of the frames has no pictures-in-picture, one third has one picture-in-picture, and one third has two pictures-in-picture.
4. Tups-reconf. Half of the generated frames is generated in copy-mode. For the other half motion-estimation mode is used.

All reconfigurations are matched by their inverse to cancel out latency differences. Because we are interested in average reconfiguration latency, multiple reconfiguration pairs are performed at regular intervals. The PiP variants perform eight reconfigurations in total. Tups-reconf performs four reconfigurations.

Overhead. To measure reconfigurability overhead, we compared the run time of these applications to the run time of equivalent static applications. For example, the run time of PiP-01 is compared to the average run time of PiP-0 and PiP-1.

Figure 7 shows the overhead of the reconfigurable applications using one to eight processors. We have omitted measurements at 9 processors due to the result of PiP-1 at 9 processors. (PiP-1 does not scale beyond 8 processors, and all PiP overheads are (partly) based on the performance of PiP-1.) An overhead factor of 1.01 means the reconfigurable program is 1 percent slower than the corresponding static applications.

Although reconfiguration occurs very often (once every 12 frames for the PiP applications, once every 17 frame for Tups-reconf), the overhead is at most 8 %. When the application is stopped for reconfiguration, the amount of parallelism in the application drops until the application is run sequentially. Thus, on average there is less parallelism to exploit in the reconfigurable applications and the reconfigurable applications will perform relatively worse on larger numbers of nodes. This causes the reconfigurability overhead to increase with the number of nodes, which is is clearly visible in Fig. 7.

Latency. We have measured the latency of reconfigurations. This is the time between the occurrence of the asynchronous event and the completion of the reconfiguration. The latency includes waiting until the application is idle and performing the reconfiguration.

Figure 8 shows the average reconfiguration latency. Because anomalies occur at 1 and 2 nodes, we have not included measurements at 1 and 2 nodes. These anomalies occur when the event is generated just before or after many computations. For example, in a sequential application it can happen that all active iterations are about to start the manager (see Fig. 4). When an event occurs at this point, there are no computation jobs in the job queue (only small manager jobs) and the application quickly becomes idle. On the other hand, latency will be high if many computation jobs have been scheduled when the event occurs. When the applications are run at higher number of nodes this effect becomes less visible and the average latency goes to 40 ms which is a single image frame in a 25 Hz video stream.

The individual latency measurements show that the latency depends on the complexity of the program before reconfiguration. This is because the complexity determines the (average) number of outstanding computations and thereby the time before the application is idle. For example, in the Tups-reconf application, the latency of going from copy-mode to motion-estimation mode is 10 ms (at 3 or more nodes). The latency of going from motion-estimation mode to copy-mode varies between 56 and 85 ms at 3 or more nodes.

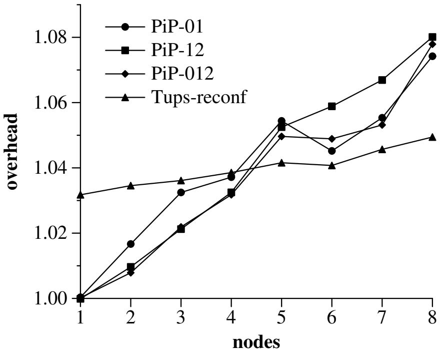

Fig. 7. Reconfigurability overhead

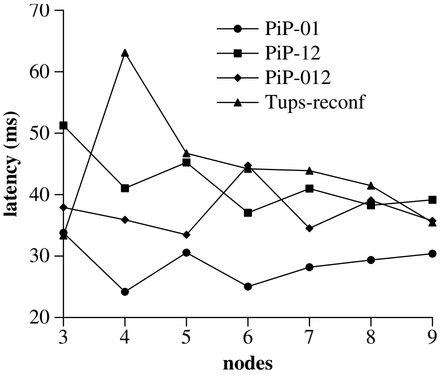

Fig. 8. Reconfigurability latency

5 Related Work

There are many other systems that simplify programming multimedia applications for embedded architectures by providing abstractions. Often these systems include hardware design. Some systems mainly focus on hardware design, like Cheops[12] and Imagine[13].

At Philips Research many of these systems have been developed, e.g., TTL[14], YAPI[15] and C-HEAP[16]. These systems model an application as a Kahn Process

Network (KPN)[17], which is a number of independent tasks that communicate using FIFO channels. These tasks and FIFO channels can be implemented in hardware or software, using shared or distributed memory. Task parallelism is supported this way, however, data parallelism is not. Load balancing is mostly done statically by mapping the tasks to fixed resources. C-HEAP has the most advanced reconfiguration support of these systems.

The SmartCam project[18] aims at simplifying building image processing applications for use in smart camera's. Skeletons[19] are used to provide an implementation abstraction and to exploit data parallelism. Tasks can be mapped on different hardware in different (heterogeneous) hardware architectures. Memory is distributed in these architectures. Unlike Hinch, SmartCam does not do reconfiguration or dynamic task scheduling. User events are absent in the SmartCam application domain.

The Model Integrated Real-Time Imagine Processing System[20] is a programming environment for building image processing applications, which are run on a network of DSPs. MIRTIS generates a parallel image processing application from sequential kernel code, data dependencies and the application graph. The generated application includes a run time system for a distributed computing platform. MIRTIS supports both data and task parallelism. The mapping of the application to the parallel hardware is done statically. Dynamic load balancing and runtime reconfiguration are therefore not supported. User events do not occur in the applications MIRTIS supports. MIRTIS deliberately does not have support for (designing) specialized hardware.

The Nizza framework[21] has a similar structure as Hinch. It also processes streaming multimedia applications in dataflow-style, allowing task parallelism. Data parallelism can be exploited using 'combinatorial' modules. The application can stop Nizza if it wants to perform reconfiguration and restart Nizza afterwards. Unlike Hinch, Nizza targets desktop applications instead of embedded applications. Therefore, Nizza does not have support for (designing) specialized hardware and distributed memory. To our knowledge, Nizza does not support handling user events.

Various projects are dealing with programming SoC architectures in the domain of network processing. Among these projects are NP-Click[22], NEPAL[23], Shangri-La[24], the system described in [25], and Netbind [26]. Like multimedia applications, network processing applications also process streams of data (network packets) by multiple kernels. However, these kernels are much smaller than multimedia kernels. To exploit this fine grained parallelism, the systems cooperate closely with the hardware.

Table 1 provides a summary of the features of the mentioned multimedia programming systems and Hinch. The first two columns indicate the presence of support for task- and data-parallelism, respectively. The load balancing column indicates the quality of the load balancing features of the system. The Dist. mem column indicates if the system targets distributed memory architectures. A '-' in this column means the system targets shared memory architectures. The events column indicates if the system has support for handling asynchronous user events. Finally, the hardware column shows if the system has support for specialized acceleration hardware in the target architecture. Table 1 shows that distributed memory and special hardware are currently not supported by Hinch. We plan to include support for this in the future.

Table 1. Comparison of related work. '+' = excellent support, 'o' = supported, '-' = bad / no support, 'N/A' = not applicable

Feature	Task par.	Data par.	Load balancing	Dist. mem	Reconfigurability	Events	Hardware
C-HEAP	+	-	o	o	+	o	o
SmartCam	+	+	o	o	-	N/A	o
MIRTIS	+	+	o	o	-	N/A	-
Nizza	+	o	+	-	o	-	-
Hinch	+	+	+	-	+	+	-

6 Conclusion

We have presented Hinch, a runtime system for multimedia applications. Hinch has support for streaming, event handling, reconfiguration, data parallelism and task parallelism, amongst others. Hinch also provides automatic load balancing of the application when run on a shared-memory architecture, by running the application in a data-flow style. Experiments show that applications using Hinch can be efficiently parallelized and the overhead of reconfigurating a running program is low.

Future work includes building a layer on top of Hinch that provides a simple interface for specifying multimedia applications. This layer will automatically generate optimized applications that use Hinch. We plan to include performance prediction in this layer. Other future work tracks are adding support for specialized hardware and more complex memory architectures, such as multiple SoC tiles.

Acknowledgments

We would like to thank Philips Research, especially Paul Stravers, for their support. This work is performed in the context of STW/PROGRESS project DES.6397, which is supported by the Dutch government.

References

1. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM Journal of Research and Development **49**(4/5) (2005) 589
2. Intel Corporation: Network processors. http://www.intel.com (2006)
3. Stravers, P., Hoogerbrugge, J.: Single chip multiprocessing for consumer electronics. In Bhattacharyya, ed.: Domain-Specific Processors. Marcel Dekker (2003)
4. Bal, H.E., Haines, M.: Approaches for integrating task and data parallelism. IEEE Concurrency **6**(3) (1998) 74–84
5. González-Escribano, A., van Gemund, A.J., noso Payo, V.C.: An XML structured representation for nested-parallel programming languages. In: Proc. CPC, Chiemsee (2004) 149–160
6. Lee, E.A., Parks, T.M.: Dataflow process networks. In: Proc. of the IEEE. (1995) 773–799
7. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. In: Proc. PPoPP'01, Snowbird, UT (2001)
8. Varbanescu, A.L., Sips, H., van Gemund, A.: PAM-SoC: A toolchain for predicting MPSoC performance. In: Proc. EuroPAR '06, Dresden (2006)

9. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3) (2000) 78–85
10. Thies, W., Karczmarek, M., Amarasinghe, S.P.: StreamIt: A language for streaming applications. In: Proc. CC 2002, Grenoble, France (2002) 179–196
11. Ramachandran, U., Nikhil, R.S., Harel, N., Rehg, J.M., Knobe, K.: Space-time memory: A parallel programming abstraction for interactive multimedia applications. In: Proc. PPoPP'99, Atlanta, Georgia (1999) 183–192
12. Bove, V., Watlington, J.: Cheops: A reconfigurable data-flow system for video processing. IEEE Trans. on Circuits and Systems for Video Technology **5**(2) (1995) 140–149
13. Serebrin, B., Owens, J.D., Chen, C.H., Crago, S.P., Kapasi, U.J., Khailany, B., Mattson, P., Namkoong, J., Rixner, S., Dally, W.D.: A stream processor development platform. In: Proc. 20th International Conference on Computer Design, Freiburg, Germany (2002)
14. van der Wolf, P., de Kock, E., Henriksson, T., Kruijtzer, W., Essink, G.: Design and programming of embedded multiprocessors: an interface-centric approach. In: Proc. CODES+ISSS. (2004) 206–217
15. de Kock, E.A., Smits, W.J.M., van der Wolf, P., Brunel, J.Y., Kruijtzer, W.M., Lieverse, P., Vissers, K.A., Essink, G.: Yapi: application modeling for signal processing systems. In: Proc. 37th Design Automation Conference, New York, NY, USA, ACM Press (2000) 402–405
16. Nieuwland, A., Kang, J., Gangwal, O.P., Sethuraman, R., Busá, N., Goossens, K., Llopis, R.P., Lippens, P.: C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. Design Automation for Embedded Systems **7**(3) (2002) 233–270
17. Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, J.L., ed.: Information processing, Stockholm, Sweden, North Holland, Amsterdam (1974) 471–475
18. Caarls, W., Jonker, P., Corporaal, H.: Skeletons and asynchronous RPC for embedded data- and task parallel image processing. In: Proc. MVA2005, Tokyo (2005) 384–387
19. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press & Pitman (1989)
20. Moore, M.S., Sztipanovitz, J., Karsai, G., Nichols, J.: A model-integrated program synthesis environment for parallel/real-time image processing. In: Proc. Par. Dist. Methods for Image Processing. (1997) 31–45
21. Tanguay, D., Gelb, D., Baker, H.H.: Nizza: A framework for developing real-time streaming multimedia applications. Technical Report HPL-2004-132, HP Labs, Palo Alto (2004)
22. Shah, N., Plishker, W., Keutzer, K.: NP-Click: A programming model for the Intel IXP1200. In: Proc. NP-2 conjunction with HPCA-9, Anaheim, California (2003)
23. Memik, G., Mangione-Smith, W.: Nepal: A framework for efficiently structuring applications for network processors. In: Proc. NP-2 in conjunction with HPCA-9, Anaheim, California (2003)
24. Chen, M.K., Li, X.F., Lian, R., Lin, J.H., Liu, L., Liu, T., Ju, R.: Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. In: Proc. ACM SIGPLAN PLDI, New York, NY, USA, ACM Press (2005) 224–236
25. Ramaswamy, R., Weng, N., Wolf, T.: Application analysis and resource mapping for heterogeneous network processor architectures. In: Proc. NP-3 in conjunction with HPCA-10, Madrid, Spain (2004) 103–119
26. Campbell, A.T., Chou, S.T., Kounavis, M.E., Stachtos, V.D., Vincente, J.: Netbind: A binding tool for constructing data paths in network processor-based routers. In: Proc. IEEE OPENARCH, New York, NY (2002)